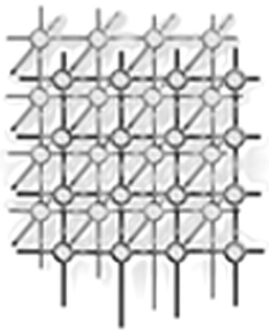
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–27 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Comparison of GPU Architectures for Asynchronous Communication with Finite-Differencing Applications



D.P. Playne*and K.A. Hawick†

Institute of Information and Mathematical Sciences, Massey University – Albany, North Shore 102-904, Auckland, New Zealand.

SUMMARY

Graphical Processing Units (GPUs) are good data-parallel performance accelerators for solving regular mesh partial differential equations (PDEs) whereby low-latency communications and high compute to communications ratios can yield very high levels of computational efficiency. Finite-difference time-domain methods still play an important role for many PDE applications. Iterative multi-grid and multilevel algorithms can converge faster than ordinary finite difference methods but can be much more difficult to parallelise with GPU memory constraints. We report on some practical algorithmic and data layout approaches and on performance data on a range of GPUs with CUDA. We focus on the use of multiple GPU devices with a single CPU host and the asynchronous CPU/GPU communications issues involved. We obtain more than two orders of magnitude of speedup over a comparable CPU core.

KEY WORDS: finite-difference; GPU; multiple device; CUDA.

1. INTRODUCTION

Graphical Processing Units (GPUs) have given data-parallel computing a new lease of life through their highly-efficient tightly-coupled computational capabilities within a single compute device. While further improvement in chip fabrication techniques will undoubtedly lead to yet more cores on a single device it is already also quite feasible to consider application programs that make use of multiple devices. This architectural notion of a CPU serviced by a small to medium number of separate GPU or similar accelerator devices is a particularly attractive way of increasing the power of individual nodes of a cluster computer or supercomputer [1] using off-the shelf technological components.

*Correspondence to: IIMS Massey University – Albany, North Shore 102-904, Auckland, NZ.
†E-mail: {k.a.hawick, d.p.playne}@massey.ac.nz

Received 1 September 2010
Revised 1 September 2010

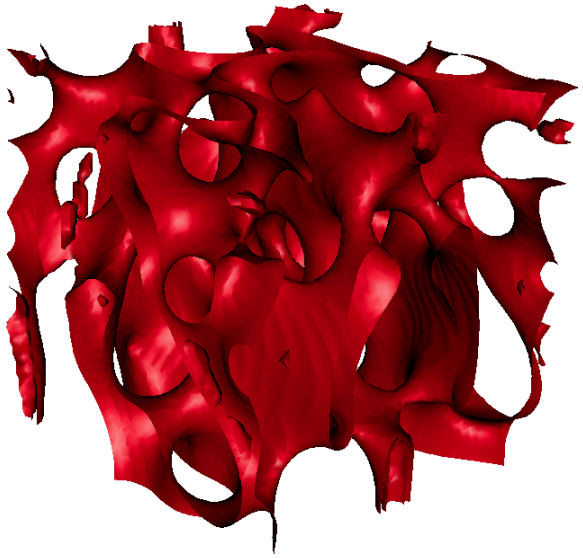


Figure 1. A sample visualisation of a three-dimensional Cahn-Hilliard finite-differencing simulation.

In this paper we look at software and ideas for managing multiple GPUs from a single CPU host – that may itself have one or more likely several conventional cores. We look at the asynchronicity issues and latency hiding potential for employing multiple host level threads within a single application, whereby each host thread runs a separate GPU device. We use and compare a trio of NVIDIA's GTX260 GPU device cards and we extend the work reported in [2] with performance data and analysis on a pair of GTX480 device cards.

A classic application problem in computational science [3, 4] is that of regular-structured hypercubic mesh [5] of floating-point field values upon which a partial differential equation (PDE) is solved iteratively using spatial stencils which provide discretised finite-difference operators.

Key issues that arise are how to arrange the problem data domain in the memory of the computational device(s) to minimise overall communications of cell halos or where to asynchronously overlap communications with computations [6]. Work has been reported elsewhere on these issues for a single host thread and a single GPU accelerator device [7]. In this present paper we focus on multiple device effects and the particular problems that arise from yet another hierarchical layer in memory access speed.

The use of GPUs to accelerate this sort of regular finite-difference calculation is not new. Researchers reported obtaining GPU speedups (over single CPU core) of around factors of eight [8] to twenty [9] on GPUs of vintage 2003 on similar problems to the ones we discuss and even for more sophisticated solver-methods such as the method of moments [10] where the data structures are harder to map to the GPU. Finite-difference method speedup on GPUs has been reported more recently between 2009 and

2010 as around a factor of forty for application problems in fluid dynamics [11] and in computational electromagnetics [12]. There have however been very few reported works on use of **multiple GPU devices** hosted by a single CPU. Egloff reports obtaining a speedup of around twenty-five with a single GPU and a speedup of around forty with two devices using a tridiagonal solver approach [13]. Our experience has been that performance speedups of between one hundred and two hundred **are possible** – but these do require detailed and programmer-intensive optimisations to make use of GPU memory layout and also the multiple device management ideas that we present in this current paper.

Halo issues in memory [14] and stencil-based problems [15] are also not new and have been widely experimented upon for many different software architectures and problems in parallel computing. Many applications in science and engineering can be formulated in this way [4] and can therefore make use of cluster supercomputers with nodes that have been accelerated with multiple devices. Applications range from those that involve simple stencil problems [16] in financial modelling [17] and in fluid mechanics [18], through complex stencils coding involving non-trivial data types such as vectors or complex numbers [19, 20], to those problems that couple multiple simple models together such as weather [21], environmental [22] and climate simulation codes [23].

We focus on regular-structured mesh with direct storage, but these ideas relate to other problems that can be formulated in terms of sparse matrices [24]. We illustrate our approach and results using code implemented in NVIDIA's Compute Unified Device Architecture (CUDA) – which is a C-based syntactic language highly suited to NVIDIA's Tesla GPU devices [25]. Our host code running on conventional CPU core(s) employs straight C code and the pThreads library for multi-core functionality [26].

This approach targets applications that can be decomposed into **asynchronously-coupled** individual computational kernels that are themselves intrinsically tightly coupled data-parallel computations. The latency balance points are unusual however because of the memory decompositions possible between the CPU main memory and the various sorts of memory available on the GPU devices themselves.

In Section 4 we summarise the GPU architecture and describe the application problem in Section 5. In Section 7 we present some performance results which are discussed in terms of the various asynchronous tradeoff possibilities and tradeoffs possible in Section 8 where we also offer some conclusions and areas for further work experimenting with asynchronous support devices and in particular for building clusters with **multiple GPU devices** on each node.

2. FINITE DIFFERENCE EQUATIONS

Finite-difference methods for solving partial differential equations on regular structured meshes generally lend themselves well to implementation on GPUs with CUDA [16]. The general method involves approximating spatial calculus operators such as derivatives

$$\frac{\partial}{\partial x}\cdot, \frac{\partial}{\partial y}\cdot, \frac{\partial}{\partial z}\cdot, \ldots \tag{1}$$

or the Laplacian

$$\nabla^2 \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\cdot, \ldots \tag{2}$$

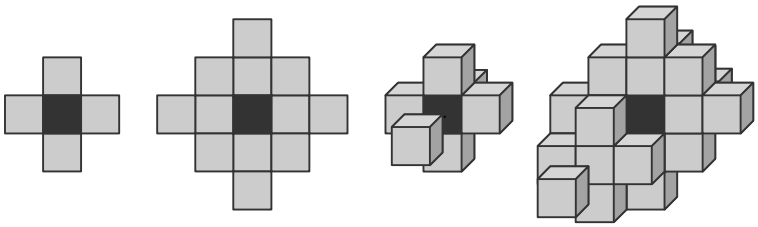


Figure 2. Some common memory halos in two and three dimensions. The light grey cells denote cells that would be used in the calculation of the finite-difference equation.

with explicit difference formulae discretised on the spatial (regular) mesh. So for example (in two dimensions) the Laplacian can be approximated by:

$$\frac{1}{\Delta^2}\left(u_{x,y,z-1} + u_{x,y-1,z} + u_{x-1,y,z} - 6u_{x,y,z} + u_{x+1,y,z} + u_{x,y+1,z} + u_{x,y,z+1}\right) \tag{3}$$

where Δ is the spatial distance between cells. This direct method extends to some quite complicated operators such as the biharmonic operator ($\nabla^4\cdot$), and other situations where operators can be composited or otherwise combined [20]. Equations of the general form:

$$\frac{u(\mathbf{r},t)}{dt} = \mathcal{F}(u,\mathbf{r},t) \tag{4}$$

where the time dependence is a first derivative and the spatial dependence in the right hand side is expressed in terms of an additive list of partial spatial derivatives, can be time-integrated using finite-difference methods. Although for explanatory purposes it is often easiest to use the first-order Euler time-integration method, in practice it is straightforward to apply other second-, fourth- or higher -order methods such as Runge-Kutta [27], once the spatial operators can be coded.

3. Memory Halo

Field equations using finite-differencing time-domain solvers almost always access the values of neighbouring cells to determine how to change the value of a cell. The neighbours that the cell must access is known as the "memory halo" of the equation. Figure 2 shows some common memory halo patterns.

The size of an equation memory halo has an effect on how well the equation can be split across multiple GPUs. When a field is split, the cells on the edge of each sub-field must be communicated to the other GPU for each step of the simulation. An equation with a large memory halo will require more cells to be communicated each step than an equation with a small memory halo. We report on work with the Cahn-Hilliard equation which has a $\nabla^4 = \nabla^2\cdot\nabla^2$ compounded Laplacian and therefore a relatively large memory halo. In this paper we discuss direct solvers using regular mesh finite-differences which give rise to banded sparse matrices. The more elaborate the memory halo, the larger the number of

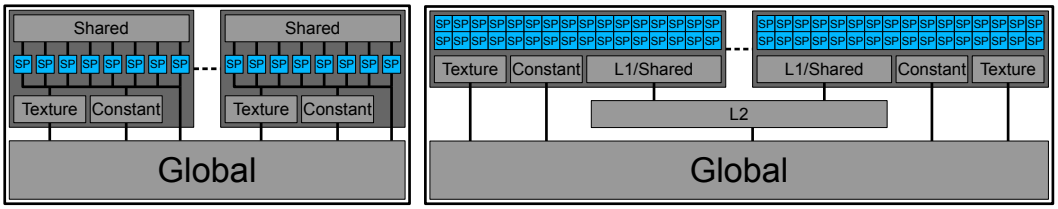


Figure 3. The architecture of the Tesla GPU (left) and Fermi GPU (right). Each GPU contains several multiprocessors, which have access to Global Memory, and on-chip Shared, Texture and Constant memory. Access to Global memory on the Fermi architecture is cached by an on-chip L1 cache and a shared L2 cache. The Tesla GPUs contain 8 scalar processors in each multiprocessor while the Fermi GPUs contain 32.

bands in the matrix [28]. Results showing the impact of the memory halo size on our performance results are presented in Section 7.

4. GPU ARCHITECTURE

Graphical Processing Units or GPU development has been driven by the increasing demands of the games market. To meet these demands, GPUs have developed into highly-parallel, many-core architectures. NVIDIA GPUs contain a scalable array of multiprocessors (MPs) [29] each of which contains scalar processors (SPs). On older generation, Tesla architecture GPUs there are 8 SPs per multiprocessor while the new Fermi architecture GPU contain 32 SPs per multiprocessor.

The programming model of CUDA GPUs is Single Instruction Multiple Thread or SIMT, this paradigm is similar to the well-known SIMD. All the SPs within the same multiprocessor must execute the same instruction synchronously, however different multiprocessors can execute instructions independently. GPUs are capable of managing thousands of threads (known in CUDA as kernels) which are scheduled and executed on the multiprocessors. As this thread management is performed in hardware, it presents little overhead.

The main performance factor that must be considered when designing a CUDA program is the memory access. GPUs are designed to have a very high computational throughput, they do not have the high-levels of memory caching common to CPUs. The Tesla architecture GPUs contain a number of specialised memory caches that must be explicitly used. The release of the Fermi architecture has also relaxed this restriction with the inclusion of L1/L2 cache on the GPU.

Global Memory is the main device memory that must be used for all GPU programs. This is the only memory that can be written to or read from by the host. This is also the slowest memory to access from the multi-processors. Access times to this type of memory can be improved, when 16 sequential threads access 16 sequential and aligned memory address, the transactions will be coalesced into a single transaction.

Shared Memory is shared between threads executing on the same multi-processor. Threads can read and write to this memory and share common information between threads.

Texture Memory is not a separate memory type but rather a cached method of accessing global memory. This cache will automatically cache values spatially surrounding those accessed. This spatiality can be defined in 1-, 2- or 3-dimensions. This memory type is best used when neighbouring threads access memory addresses in the same area.

Constant Memory is like texture memory in that it is also a cached method of accessing global memory. Instead of caching neighbouring values, constant memory cache allows all threads in a multi-processor to access the same memory location in a single transaction.

L1/L2 Cache is only available on the Fermi-architecture GPUs but significantly improves the ease of programming for the developer. L2 cache is shared between all the multiprocessors and L1 is shared between the SPs on each multiprocessor. L1 cache is stored in the same memory space as Shared memory, the user can configure whether to split this memory between L1/Shared as either 16KB/48KB or 48KB/16KB [29].

Proper use of memory and access patterns can greatly improve the performance of GPU programs. In the following section we describe how to implement a field-equation simulation across multiple GPU devices.

5. FIELD-EQUATION SIMULATION ON GPUS

In previous work [7] we have described implementations for field-equations using finite-differencing on GPUs with CUDA. The field equation used as an example is the Cahn-Hilliard equation [30], we use the same example equation for this paper. The equation is as follows:

$$\frac{\partial u}{\partial t} = \mathrm{m}\nabla^2\left(-\mathrm{b}u + \mathrm{U}u^3 - \mathrm{K}\nabla^2 u\right) \tag{5}$$

Where m, b, u, K are adjustable parameters and the d-dimensional field variable u is usually initialised randomly and updated at each time-step using the finite-difference formulation. Figure 1 shows a visualisation of an example three-dimensional Cahn-Hilliard simulation after it has been allowed to quench from this initial random state.

The basic design of the single-GPU implementation is as follows. Each cell has one kernel allocated to it, this kernel will read the value of the cell and surrounding neighbours from memory, compute the new value of this cell according to the equation and write the result to an output array.

In [7] we showed that the optimal memory type for field-equation simulations on Tesla architecture GPUs was **texture** memory. The texture memory caches values in the same spatial locality and improves on simple global memory access. We have found that this no longer holds true on the Fermi architecture GPUs. The performance of the **L1/L2** cache provides better access times than **texture** memory. The performance results we present in Section 7 use **texture** memory for the Tesla GPUs and **L1/L2** cached **global** memory for the Fermi GPUs.

In this we aim to improve this simulation by spreading the computation across multiple GPUs hosted on the same machine. We give an example of how the field can be split across two or more GPUs. First we must discuss how the field is split between the GPUs.

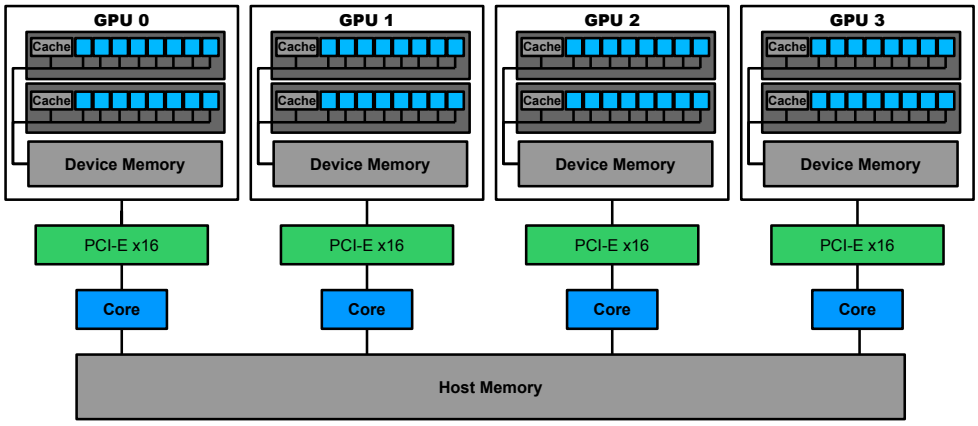


Figure 4. A multi-core CPU connecting to four graphics cards, each hosted on a separate PCI-E x16 slot.

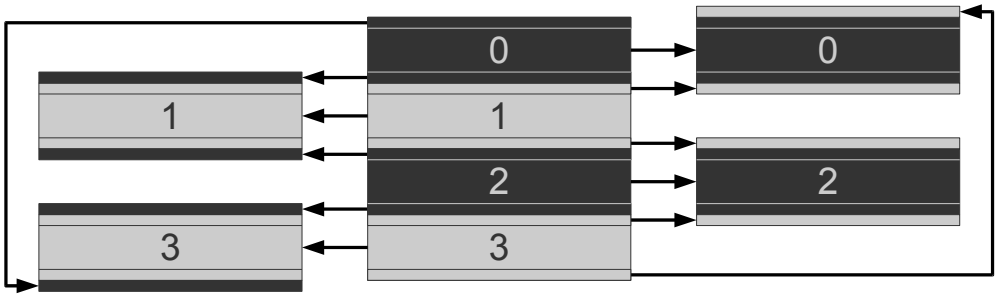


Figure 5. The main field (centre) being split into four separate fields with the appropriate border cells.

6. FIELD DECOMPOSITION

To decompose a field-equation simulation across N GPUs we must split the field into N even sections. Each section is loaded into memory on a different GPU and with two sets of bordering cells from the neighbouring sections – as shown in Figure 5. These borders are the cells from the neighbouring sections of the field that are required by the cells on the borders. The width of these borders depend on the memory halo of the model.

The field should be split in the highest-dimension, this means that the cells in the borders will be in sequential addresses. This allows the copy procedures to operate with maximum efficiency. If the field is split in the lower dimensions, multiple copy calls will be required to copy the data between the devices. These multiple copy calls will perform slower than one larger call.

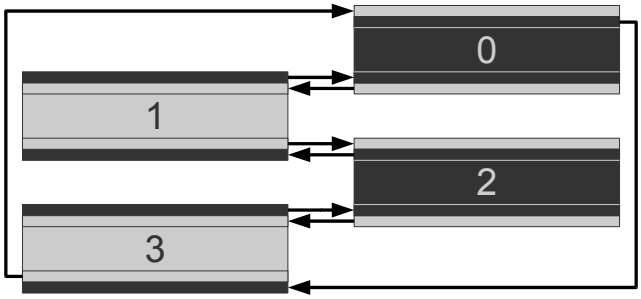


Figure 6. The communication of border cells for a field split between four GPUs, this process must be performed after each simulation time-step.

6.1. Synchronous Memory Copy

This is the simplest method of decomposing the field equation simulation across N GPUs. To interact with N GPUs, N CPU threads are required. For these implementations we use the pThreads library to manage the multi-threading on the CPU. For this implementation N threads are created to interact with one GPU each. For every step, each thread performs the following simple algorithm:

1. Compute the simulation for the cells in the field
2. Synchronise with the other threads
3. Copy bordering cells from GPU
4. Exchange bordering cells with neighbours
5. Load new bordering cells into GPU
6. Repeat

This is illustrated in Figure 6.

This method is simple to implement and provides a way to quickly utilise multiple GPUs. However, the downside of this approach is that the each GPU must calculate the simulation for the field then stop and sit idle while the CPU threads exchange data. When the field length and memory halo are both small, this memory exchange time does not take long and is does not have a large impact on performance. However, as the field length and/or memory halo increases in size, this memory exchange has a larger impact on performance.

Any time the GPU sits idle is a waste of resources, to minimise this idle-time we make use of the asynchronous memory copy/execution capabilities of the GPU.

6.2. Asynchronous Memory Copy

This implementation uses asynchronous memory access to reduce idle GPU time. CUDA supports asynchronous host-device memory access and execution for GPUs with compute capability 1.1 or higher can split device execution and memory copies into separate streams. Execution in one stream

can be performed at the same time as a memory copy from another. This can be extremely useful for reducing GPU idle time.

The basic idea behind this implementation is to use asynchronous copies to exchange the border cell values between the threads while the GPU is still computing the simulation for the rest of the field. We decompose the field into N parts as described in Section 6. Each thread in the asynchronous memory copy implementation performs the following algorithm:

1. Compute the border cells (Stream 1)
2. Compute the remaining cells in the field (Stream 2)
3. Copy bordering cells from GPU to the host (Stream1)
4. Exchange bordering cells with neighbours (CPU)
5. Load new bordering cells into GPU (Stream 1)
6. Repeat

The implementation of this algorithm is shown in Listing 1.

Listing 1. Implementation of the Asynchronous Copy multi-GPU implementation.

```
int idm1 = (id == 0) ? NUM_THREADS-1 : id-1;
int idp1 = (id == NUM_THREADS-1) ? 0 : id+1;

dim3 b_threads(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 b_grid(X/threads.x, 2);

dim3 f_threads(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 f_grid( X/threads.x, (Y2 / threads.y) - 2 );

for(int i = 0; i < no_steps; i++) {
    compute_border<<<b_grid, b_threads,0,stream1>>>(d0,d1);

    cudaMemcpyAsync(&swap[id][0],    &d1[h*X],   h*X*sizeof(float),
                    cudaMemcpyDeviceToHost, stream1);
    cudaMemcpyAsync(&swap[id][h*X], &d1[Y2*X], h*X*sizeof(float),
                    cudaMemcpyDeviceToHost, stream1);

    compute_field<<<f_grid, f_threads,0,stream2>>>(d0,d1);

    cudaStreamSynchronize(stream1);

    sem_post(&copy_m1[id]);
    sem_post(&copy_m1[id]);

    sem_wait(&copy_m1[idm1]);
    sem_wait(&copy_p1[idp1]);

    cudaMemcpyAsync(&d1[(Y2+h)*X], &swap[idp1][0],    h*X*sizeof(float),
                    cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(&d1[0],          &swap[idm1][h*X], h*X*sizeof(float),
                    cudaMemcpyHostToDevice, stream1);
```

```
    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2);
}
```

There are some programatic challenges that must be overcome when implementing this design that are worth noting. For the GPU to copy data from the device memory to the host memory asynchronously, the host memory must be allocated by CUDA to ensure that it is page locked. This is normally performed using the CUDA function `cudaMallocHost( void **ptr, size_t size)`. However, memory declared using this function will only be accessible to the thread that declared it. This means that exchanging the bordering cells requires an extra CPU memory copy to copy the data between the page-locked memory for each thread. This extra memory copy is obviously undesired.

This problem can be overcome by allocating the memory using the function `cudaHostAlloc(void **ptr, size_t size, unsigned int flags)` with the flag `cudaHostAllocPortable`. This flag tells the compile to make the memory available to all CUDA contexts rather than only the one used to declare it. In this way both threads can use the memory to exchange border cells.

7. RESULTS

To test the multi-GPU designs for field equation simulations, we have implemented and compared a sample field equation on both architecture. We selected the Cahn-Hilliard equation because we have previously reported on its single-GPU performance and believe it to be a good representative for finite-differencing simulations. We have implemented this simulation using both multi-GPU method and compared their performance on both GPU architectures to previously recorded single-GPU results. All implementations make use of the Euler integration method, this method is not used for actual simulations due to its poor accuracy, however for ease of description and performance comparison it is suitable.

The details of the platform the implementations have been tested on are as follows. The Tesla machine is a 2.66GHz Intel® Core™2 Quad CPU and 8GB of DDR2-800 memory. The GPUs used for the simulations are three NVIDIA® GeForce® GTX 260s each of which contains 27 multiprocessors (216 SPs) each access to 896MB of memory. The Fermi machine is a 2.67GHz Intel® Core™ i7 920 with 6GB of DDR3-1600 memory. The Fermi GPUs are two NVIDIA® GeForce® GTX 480s which contain 15 multiprocessors (480 SPs) and each have access to 1536MB of memory. Both machines are running on Ubuntu 9.10 64-bit and are using CUDA 3.1.

The performance of both implementations and both GPU architectures have been tested across a range of field-lengths and memory halos. Equations with larger memory halos must access more values and are generally slower which distorts the performance comparison. Therefore the same equation is used but the number of bordering cells copied each iteration is increased (as appropriate to the stated memory halo). This shows a comparison of the performance of the implementation at providing the necessary data rather than a comparison between the performance of equations with varying sizes of memory halo.

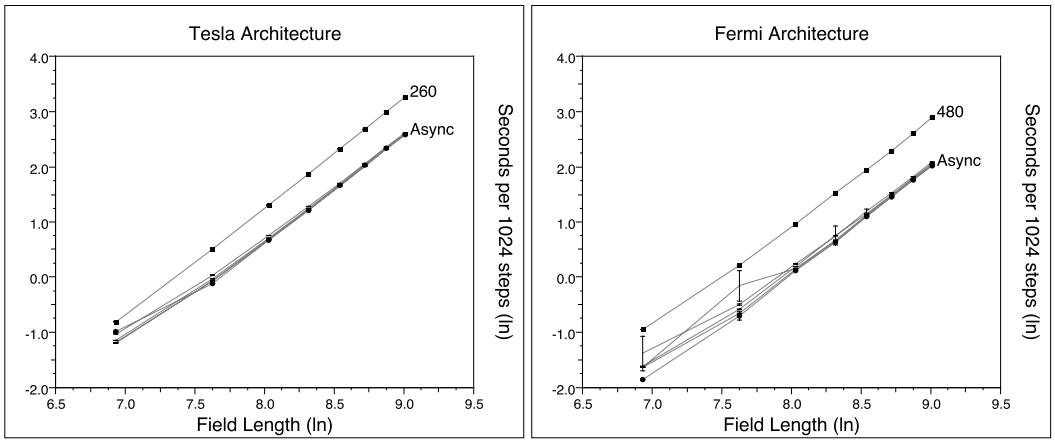


Figure 7. Performance results showing the seconds per 1024 simulation time-steps for the two multi-GPU implementations, results are shown for field lengths of $N = 1024^2...8192^2$ with memory halo sizes 2, 4, 8 and 16. Results are shown for the Tesla architecture (left) and Fermi architecture (right).

In Figure 7 shows a comparison of the multi-GPU synchronous and asynchronous copy methods for the Tesla and Fermi architectures. These results shows a diminished difference between the two multi-GPU methods using CUDA 3.1 compared to the results published in [2] using CUDA 2.3.

The CUDA 3.1 release has improved the performance of the simulations, however also exposed a multi-threading reliability issue in the Synchronous copy implementation. This necessitated the change to a more low-level multi-threading synchronisation method which has brought the Synchronous method performance closer to the Asynchronous method. The multi-threading issue was also present in the Asynchronous copy implementation but this method was robust against the effects.

At this level of highly optimised implementation, even minor issues can have a high impact on overall performance. The Asynchronous memory copy method is still valuable as it still provides slightly better performance and is not affected by the issues that the Synchronous method is susceptible too.

To see how the Tesla and Fermi architecture GPUs compare, we have shown the performance of 1, 2 and 3 GTX260s and 1 and 2 GTX480s for field lengths of $N = 1024^2...8192^2$. For the multi-GPU results we have used the Asynchronous copy method as it provides the best results. These results are shown in normal and ln-ln scale in Figure 8.

The performance results shown in Figure 8 are consistent with the expected problem complexity. It should be noted that the GTX260x3 configuration is slower than a single GTX260 for a field length of $N = 1024^2$. At this small scale the communication and startup overhead in using **three** GPUs outweighs the computation benefits.

It is interesting to note that for the Fermi architecture, both the Synchronous and Asynchronous methods provide a super-linear speedup over the single GTX480. We believe the reason for this super-linear speedup is due to the newly introduced **L1/L2** cache used by the Fermi GPUs. By processing

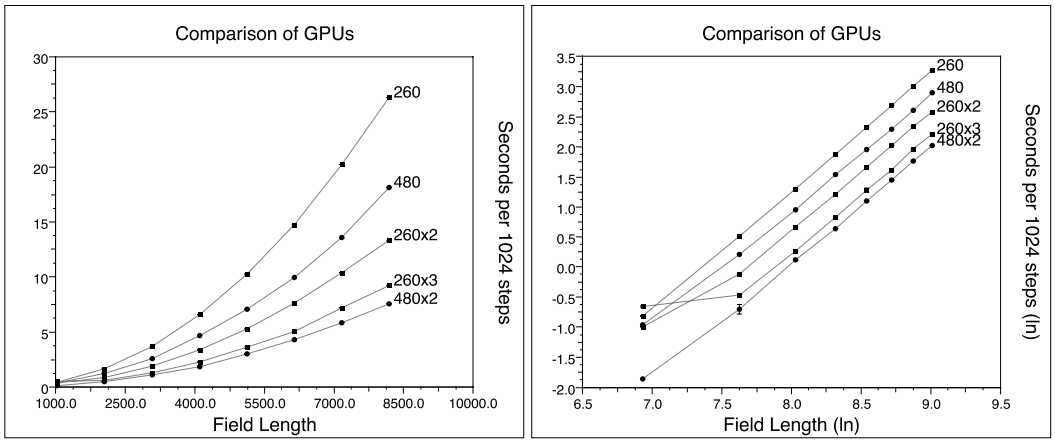


Figure 8. Comparison of GPU configurations for field lengths of $N = 1024^2 ... 8192^2$. Results shown in normal scale (left) and ln-ln scale (right). Problem complexity is $O(N = L^2)$ as expected for a two-dimensional field. Note that error-bars are shown but are smaller than the plot symbols.

Implementation	Speedup vs CPU
GTX260	63x
GTX260x2 (Sync)	120x
GTX260x2 (Async)	124x
GTX260x3 (Sync)	176x
GTX260x3 (Async)	180x
GTX480	91x
GTX480x2 (Sync)	205x
GTX480x2 (Async)	219x

Table 7. A comparison of how GPU architectures and multi-GPU implementations compare to a single-core CPU reference implementation. Data presented are accurate to better than the significant figures shown.

a field half the size, the Fermi architecture GPUs can make better use of the cache structure and thus provide more than a 2x speedup over a single card. The performance data are summarised in Table 7.

8. DISCUSSION AND CONCLUSIONS

We have investigated the potential of a single CPU host for optimally managing multiple GPU accelerator devices. This approach is likely to be of great importance in designing future computer

clusters. We have focused on finite-difference time-domain applications as our representative benchmark problem and have extended our previous work to now use three-device configurations and also to use NVIDIA's currently available device with the most cores – the GTX480.

We have presented two methods for implementing finite-differencing field-equation simulations on **multiple** GPUs. The implementations we have discussed and presented performance data on all operate on a single host machine containing two or three GPU devices. We have shown that the correct use of asynchronous memory communication can reliably provide linear speed up over a single-GPU implementation and on NVIDIA's recent Fermi architecture GPUs this speed up can sometimes even be super-linear due to this device now having cache memory. We have found that with a carefully configured system, synchronous communication can actually provide comparable performance but is very sensitive to any thread synchronisation delay effects.

While GPGPU has opened the door to cheap and powerful data-parallel architectures, these devices do not necessarily have the same scalability of cluster machines. However good scalability can be achieved by extending a single host system to being a many-GPU cluster. In such a cluster each node is then itself a highly data-parallel machine containing one or more GPU devices.

The system described in this work performs communication between devices using CPU memory, thus the communication latency is relatively low. In a multiple GPU accelerated cluster, proper use of asynchronous communication becomes even more vital to fully utilise computing resources.

For the work we report in this paper each GPU device has around 1-1.5GByte of memory. These "gamer-level" devices are considerably cheaper than the "professional-level" devices available such as the C2070 which has 6GBytes of memory. It remains an interesting tradeoff space to decide whether to configure a compute cluster with more, cheaper cards or with few, well-configured ones. We plan on experimenting with more GPUs within a single hosted machine and also examining approaches for decomposing our simulations across clusters with non-homogeneous, GPU-accelerated nodes. This will include possible methods for pipelining communications for such systems to improve compute/communication ratios.

In summary, CUDA-programmed multiple GPUs offer an attractive route for building future generation cluster nodes. This is particularly worthwhile if the more esoteric CUDA features are exploitable by the application programmer, where as we have shown, two orders of magnitude speed up over a current generation single CPU core are possible.

REFERENCES

1. Rabenseifner R, Wellein G. *Modeling, Simulation and Optimization of Complex Processes*, chap. Comparison of Parallel Programming Models on Clusters of SMP Nodes. Springer, 2005; 409–425.
2. Playne D, Hawick K. Hierarchical and Multi-level Schemes for Finite Difference Methods on GPUs. *Proc. CCGrid 2010, Melbourne, Australia*, CSTN-099, 2010.
3. Hawick KA, Bogucz EA, Degani AT, Fox GC, Robinson G. Computational Fluid Dynamics Algorithms in High-Performance Fortran. *Proc. AIAA 25th Computational Fluid Dynamics Conf*, 1995.
4. McInnes LC, Allan BA, Armstrong R, Benson SJ, Bernholdt DE, Dahlgren TL, Diachin LF, Krishnan M, Kohl JA, Larson JW, *et al.*. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chap. Parallel PDE-Based Simulations Using the Common Component Architecture. Springer, 2006; 327–381, doi:10.1007/3-540-31619-1.
5. Hawick KA, Playne DP. Hypercubic Storage Layout and Transforms in Arbitrary Dimensions using GPUs and CUDA. *Technical Report CSTN-096*, Computer Science, Massey University 2010. Accepted for and to appear in Concurrency and Computation: Practice and Experience.

6. Hayder ME, Ierotheou CS, Keyes DE. *Progress in Computer Research*, chap. Three parallel programming paradigms: comparisons on an archetypal PDE computation. ISBN:1-59033-011-0, Nova Science, 2001; 17–38.
7. Leist A, Playne D, Hawick K. Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. *Concurrency and Computation: Practice and Experience* December 2009; **21**:2400–2437, doi:10.1002/cpe.1462. CSTN-065.
8. Krakiwsky SE, Turner LE, Okoniewski MM. Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU). *IEEE MIT-S Digest* 2004; **WEIF-2**:1033–1036.
9. Kruger J, Westermann R. Linear algebra operators for gpu implementation of numerical algorithms. *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM: New York, NY, USA, 2003; 908–916, doi:http://doi.acm.org/10.1145/1201775.882363.
10. Peng S, Nie Z. Acceleration of the Method of Moments Calculations by Using Graphics Processing Units. *IEEE Trans. Antennas and Propagation* July 2008; **56**(7):2130–2133.
11. Sypek P, Dziekonski A, Mrozowski M. How to render fdtd computations more effective using a graphics accelerator. *IEEE Trans. on Magnetics* 2009; **45**:1324–1327.
12. Zainud-Deen S, El-Deen E, Ibrahim M, Awadalla K, Botros A. Electromagnetic Scattering Using GPU-Based Finite Difference Frequency Domain Method. *Prog. in Electromagnetics Res. B* 2009; **16**:351–369.
13. Egloff D. High Performance Finite Difference PDE Solvers on GPUs. *Technical Report*, QuantAlea Gmbh 2010.
14. Grote M, Simon HD. Parallel preconditioning and approximate inverses on the connection machine. *Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing*, vol. 2, Sincovec R (ed.), SIAM, 1993; 519–523.
15. Barrett RF, Roth PC, Poole SW. Finite difference stencils implemented using chapel. *Technical Report ORNL Technical Report TM-2007/122*, Oak Ridge National Laboratory 2007.
16. Micikevicius P. 3D finite difference computation on GPUs using CUDA. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ISBN:978-1-60558-517-8, 2009.
17. Bunnin FO, Guo Y, Ren Y, Darlington J. Design of high performance financial modelling environment. *Parallel Computing* 2000; **26**:601–622.
18. Tanyi BA, Thatcher RW. Iterative Solution of the Incompressible Navier-Stokes Equations on the Meiko Computing Surface. *Int. Journal for Numerical Methods in Fluids* 1996; **22**:225–240.
19. Hawick KA, Playne DP. Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation. *International Journal of Computer Aided Engineering and Technology (IJCAET)* 2010; **2**(CSTN-075):78–93.
20. Hawick K, Playne D. Automated and parallel code generation for finite-differencing stencils with arbitrary data types. *Proc. Int. Conf. Computational Science, (ICCS), Workshop on Automated Program Generation for Computational Science, Amsterdam June 2010.*, CSTN-106, 2010.
21. Vu VT, Cats G, Wolters L. Asynchronous Communication in the HIRLAM Weather Forecast Model. *Proc. 14th Annual Conference of the Advanced School for Computing and Imaging (ASCI)*, 2008.
22. Ashworth M, Foelkel F, Gulzow V, Kleese K, Eppel D, Kapitza H, Unger S. Parallelisation of the GESIMA Mesoscale Atmospheric Model. *Parallel Computing* 1997; **23**:2201–2213.
23. Mineter M, Hulton N. Parallel processing for finite-difference modelling of ice-sheets. *Computers & Geosciences* 2001; **27**:829–838.
24. Filippone A, Colajanni M. PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices. *ACM Trans. Mathemetical Software* December 2000; **26**(4):527–550.
25. Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* March-April 2008; **28**(2):39–55.
26. IEEE. *IEEE Std. 1003.1c-1995 thread extensions* 1995.
27. Shampine LF. Some practical Runge-Kutta Formulas. *Mathematics of Computation* January 1986; **46**(173):135–150. ISSN: 0025-5718.
28. Golub GH, Loan CFV. *Matrix Computations*. 3rd edn., ISBN-13: 978-0801854149, Johns Hopkins Univ. Press, 1996.
29. NVIDIA® Corporation. *CUDA™ 3.1 Programming Guide* 2010. URL http://www.nvidia.com/, last accessed August 2010.
30. Cahn JW, Hilliard JE. Free Energy of a Nonuniform System. I. Interfacial Free Energy. *The Journal of Chemical Physics* 1958; **28**(2):258–267.